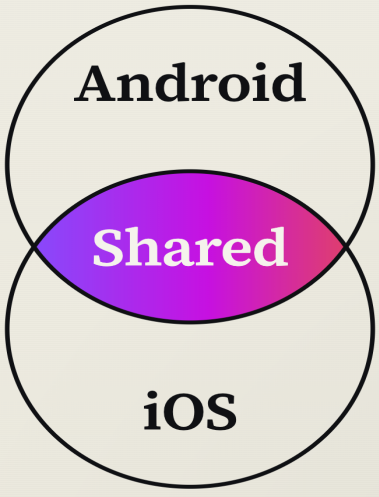
AN OPERATIONAL GUIDE

Migrating Production Apps to Kotlin Multiplatform & Compose Multiplatform

For teams shipping existing Android & iOS apps.



TWO PLATFORMS, ONE SHARED CORE

ANUBHAV SAVANT

FIRST EDITION · 2026

Contents

Copyright

© 2026 Anubhav Savant. All rights reserved.

No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Companion Code License

All code listings in the book and the accompanying companion samples repository (savantarch/kmp-migration-samples) are licensed under the **MIT License**.

First Edition: May 2026

Preface

This book is for engineering teams adopting Kotlin Multiplatform (KMP) and Compose Multiplatform (CMP) in existing production Android + iOS apps, not greenfield projects. It is *operational* in intent: the technical mechanics of KMP and CMP are covered well elsewhere; what this book covers is the *adoption work around them* — inventory, sequencing, rollout, observability, team coordination, and the anti-patterns that stall migrations.

The book spans five areas: the decision to migrate (Part I); architecture, navigation, dependency strategy, anti-patterns, and team shape (Part II); stage-by-stage walks through the Modern App sample — Compose × SwiftUI — from `stage-0` through `stage-6` (Part III); the same for the Mixed App — XML+Compose × UIKit+SwiftUI — from `stage-0` through `stage-7` (Part IV); and build integration, observability, and continuous shipping (Part V).

The JetBrains docs, Google's KMP migration guide, and Touchlab's CMP Transition Guide cover the technical mechanics well. What this book adds:

- Decision frameworks for leaders — cost models, dealbreaker checklists, phase-gate templates, team-shape patterns.
- Two UI combinations — Modern App (Compose × SwiftUI) and Mixed App (XML+Compose × UIKit+SwiftUI) run through the same 8-stage migration journey with parallel journey chapters.
- Working samples at every stage — each stage is a runnable snapshot directory on both platforms.
- Operational depth beyond the code — observability, continuous shipping, rollback design, and org shape alongside the technical patterns.

What it doesn't cover: Kotlin or Swift fundamentals, KMP/CMP API reference (JetBrains and Touchlab do that well), or CI/CD pipeline mechanics (these vary too much per organization to be useful).

Companion samples

The book is paired with a unified companion samples repository hosted on GitHub. Scan the QR code below or click the link to access the repository:

savantarch/kmp-migration-samples

Every code listing and architecture diagram in a journey chapter references a specific stage — for example, `stage-5` — so the sample and the prose stay in lockstep. If you're following along in an e-reader, clone the repository and navigate to the relevant stage directory (e.g., `modern-app/stage-5` or `mixed-app/stage-5`) to see the exact state described. Every claim that depends on the sample is testable against it.

Both samples are complete: Modern App at `stage-0` through `stage-6`; Mixed App at `stage-0` through `stage-7`. The two samples do not represent difficulty levels; they correspond to different host application architectures. Modern App uses a declarative shell (Compose and SwiftUI on both platforms, with modern navigation); Mixed App uses a heterogeneous shell (Fragment + XML NavGraph on Android, and `UINavigationController` on iOS, where declarative and pure-imperative subtrees coexist). Both apps follow the same migration path starting from different architectural baselines; neither is the "easy" or "real" track. Chapter 2 develops this distinction.

How to read this book

This book covers both deciding and doing. AI-assisted development has softened the line between "leader deciding" and "engineer doing" — most readers will find themselves crossing it. Three paths — start where you are:

- **Deciding whether to migrate** — Part I plus Chapter 1's *When not to migrate* section. Decision frameworks, dealbreaker checklist, re-evaluation cadence. Fastest path to an informed yes/no.
- **Committed and grappling with the work** — Part II (principles) plus one journey Part (III for Modern App, IV for Mixed App). The principles give vocabulary; the journey shows how they play out in the actual sample, stage by stage.
- **Implementing** — read everything. The principles in Part II are prerequisites for the operational decisions in Part V (build integration, observability, continuous shipping).

The journey chapters in Parts III and IV are self-contained — each walks through one stage (or stage cluster) with the *before* state, the migration moves, and the *after* state. Code listings reference the specific stage directory in the companion sample, so you can follow along in the repo at

any point.

Conventions

Listings longer than 20 lines are captioned as Listing N.N with a short title and a stage folder reference so you can find the file in the companion repo:

> **Listing 14.3 — `ArticlesViewModel.kt` at `stage-3`**

Shorter inline snippets need no caption. Architecture diagrams are captioned as Figure N.N and embedded in line.

Three callout types appear throughout:

Note: an observational point that adds depth without changing the argument.

Warning: a pattern, decision, or sequence that has gone wrong in practice and is worth pausing on.

Tip: a concrete, actionable suggestion the reader can apply.

Stage directories are always named so you can follow in the repo without losing your place:

> *"At `stage-5`, `AppNavHost(...)` in commonMain..."*

The `modern-app/stage-5` directory gives you the exact state described in the prose.

Part I — Foundation

The first six chapters lay the groundwork the rest of the book builds on. **Should you migrate?** quantifies the decision. **Sizing the migration** names what's actually hard about migrating an existing app (the architectural distance, not the language gap), and walks the inventory + triage practice that turns the abstract problem into a structured backlog. **The migration journey** introduces the eight stages the rest of the book is organized around — three of them terminal stages where teams can stop and ship indefinitely. **Introduction to KMP and CMP** is a quick reference for the core KMP and CMP concepts. **KMP for iOS** covers the iOS-side concerns—Swift-surface design, framework size, SKIE vs Swift export—that determine whether the iOS team champions the migration or tolerates it. **Shared UI Resources and Design Theming** establishes token-driven architecture patterns and explains Material3 theming constraints on both platforms, preparing readers for Stage 4+ CMP integration.

Readers deciding *whether* to migrate get the most out of Chapter 1; readers leading the implementation get the most out of Chapters 2 through 6.

Chapter 1 — Should you migrate?

Many organizations assume they will migrate to KMP/CMP as soon as one team starts experimenting. That is a mistake. Migration is only worth it under specific business and engineering conditions; otherwise, the costs will outweigh the benefits. This chapter gives leaders a self-contained decision framework. Its central original contributions are:

- a clear-eyed naming of the *cost of the status quo* (so the alternative is a real comparison, not a strawman),
- the *dealbreaker checklist* that says *stay native*,
- and the *asymmetric-downside argument*: even if KMP under-delivers for your organization, the investment is preserved in a way that other cross-platform technologies do not match.

The cost of the status quo

Before looking at KMP/CMP, calculate what you are already spending to maintain two parallel codebases. Eight recurring costs apply to almost every multi-team Android + iOS organization (JetBrains enumerates a related set of eleven pain points; the curated subset below emphasizes the costs that matter most for the *should-we-migrate?* decision):

- Duplicated UI — the same screens are designed, built, and tested twice. Pixel-level alignment between platforms requires explicit coordination that does not exist by default.
- Duplicated business logic — domain models, validation rules, protocol clients, feature flags, A/B-test plumbing — written and maintained in two languages. Bugs are fixed twice; sometimes only one side gets the fix.
- Inconsistent patterns over time — the two codebases drift. Each team adopts the conventions that suit its platform, and what started as a shared mental model erodes over years.
- Feature-parity struggles — one platform consistently leads, the other consistently lags. Product becomes used to “Android-first” or “iOS-first” framing as a fact of life rather than a problem.
- Doubled testing burden — every feature is QA’d on two device matrices, two OS-version matrices, two regression suites. Testing capacity, not engineering capacity, becomes the bottleneck.
- Knowledge silos across platform teams — Android and iOS engineers operate in silos, rarely

understanding each other's code or constraints. This division sets the stage for the team-shape anti-patterns in Chapter 11.
- Higher engineering cost — the only way to scale feature throughput is more headcount on each side. Cost grows linearly with feature load on each platform independently.
- Longer release cycles — a feature is "shippable" only when both platforms are. The slower platform paces the entire feature, every feature, indefinitely.

These costs compound quietly. The value of migrating is the difference between this ongoing waste and the cost of the migration itself. If your team is small, these costs are minor and migration won't pay off. If you have lived with them for years, the payback can be huge.

Is this a safe bet *now*?

Is KMP past the early-adopter risk window? The publicly available evidence — restricted to our citation pool of JetBrains, Google, and Touchlab — points one direction:

- KMP stable release — In November 2023, JetBrains officially marked Kotlin Multiplatform as production-ready, ending the "is it ready yet?" question with a date. (source)
- Doubled adoption — JetBrains' own multiplatform overview highlights that the use of cross-platform and shared-code techniques more than doubled between 2024 and 2025. (source)
- Developer ecosystem survey — JetBrains Developer Ecosystem Survey (2024 $\rightarrow$ 2025) showed KMP usage among multiplatform-mobile respondents grew from 7% to 18% — a +157% year-over-year change. Competing approaches in the same survey window declined or stagnated.
- Google I/O 2024 — Google formally recommended KMP for sharing business logic across mobile platforms. Multiple Jetpack libraries (Room, DataStore, Lifecycle) added KMP support; Compose Multiplatform reached stable for iOS, Android, and Desktop.
- Touchlab support — Touchlab continues to publish production-oriented guidance and tooling (KMMBridge, SKIE), and has done so for several years — signaling sustained vendor investment.
- Case-study index — JetBrains maintains a curated index of multiplatform Android + iOS case studies for application-specific patterns. This book doesn't cite individual entries — their durability as references isn't guaranteed — but the index itself, alongside the official guidance and ecosystem stats above, is sufficient grounding for the "safe to bet on now" question.

When migration delivers value

Migration is most likely to pay back when *all* of the following are true:

- You operate two parallel codebases with active development and ongoing release pressure on both.

- You have observable cross-platform feature drift — the same feature ships materially differently on the two platforms, or drifts in behavior over time.
- You have duplicated business-logic bugs — the same defect class has appeared on both platforms, separately, more than once.
- You can identify a shared-code wedge — a domain (auth, persistence, networking, validation, threat detection, payments) where shared code would meaningfully reduce drift.
- Your engineering organization can absorb the learning curve without sacrificing release cadence elsewhere.
- The critical SDKs on your roadmap either have KMP support, or you have a wrapper budget for the ones that do not.

Each of these is a "yes" or "no" — partial yeses count as no. If you cannot say yes to all six, look hard at the dealbreaker list before committing.

JetBrains frames the same decision differently and our criteria are consistent with theirs. They name three conditions that favor cross-platform: the app needs to ship on both Android and iOS; you want to optimize development time; you want a single codebase for app logic while keeping full control over UI elements. Their mirrored criteria for staying native — *single-platform targeting*, *UI is critical to the product*, *team highly skilled and time-pressured* — map directly onto the dealbreakers later in this chapter.

The cost model

These costs are real and manageable, but don't downplay them when selling the migration to stakeholders:

- iOS upskilling — Historically, iOS engineers needed weeks or months to get comfortable writing shared Kotlin code. AI coding assistants speed this up, helping them become productive much faster. Deep fluency still requires sustained exposure across the migration's lifecycle; plan for both.
- iOS binary-size overhead — The Kotlin/Native runtime adds binary weight to the iOS app. Real, measurable, mitigatable; see Chapter 5 for measurement and mitigation.
- Build-configuration complexity — Integrating Gradle into the Xcode build is non-trivial, especially for multi-target / multi-scheme iOS apps.
- Distribution mechanics — Publishing the iOS framework via KMMBridge, SPM, or an internal artifact channel is its own workstream.
- Rollout timeline — A phased rollout that integrates meaningful shared code spans multiple release cycles. Most of the calendar time is consumed by quality stage gates, release-cadence coordination, testing windows, and developer upskilling — not by raw coding effort. Although AI coding assistants speed up the implementation portion, the structural pacing requirements of shipping safe, incremental releases are much harder to compress. See Chapter 3.
- Coordination overhead — Two teams that have never shared code now must — see Chapter

11. The administrative coordination is, in published experience, a more common stalling factor than any technical challenge. AI does not compress this cost.

These costs are part of what you are buying with the migration. None of them is a dealbreaker; each must be planned for explicitly.

AI-assisted development as a baseline

This book assumes AI-assisted development is the default. Every cost, timeline, and learning-curve figure is written for teams using modern coding assistants as a baseline tool — not a specialty for a small inner circle. Migrations that resist AI tooling will run slower than the figures suggest, and the gap is widening.

This matters as a leadership concern, not just an engineering detail: AI changes the *shape* of the cost curve. Some costs fall sharply; others fall not at all. The mix has direct implications for how to budget, staff, and sequence the migration.

Where AI helps most

- Mechanical porting — AI is great at repetitive translation tasks—like porting repositories, validation rules, use cases, or DTOs from Swift or Kotlin to shared `commonMain` Kotlin. The same applies to generating `expect/actual` skeletons around no-KMP SDKs (Chapter 8) and scaffolding tests for shared modules.
- Cross-language reading — Before AI assistants, having iOS engineers review Kotlin (or Android engineers review Swift) was slow and frustrating. It dragged down PR times and wore reviewers out. With AI as a reading aid, navigating unfamiliar code is much smoother. The friction here was never primarily about *writing* the other side's language — it was about *reading* it, and AI helps lower that barrier.
- Inventory and triage — The work — auditing modules, classifying SDKs by KMP support, identifying parallel-codebase feature drift — is structured analysis that AI can help compress.
- Build and CI configuration — Gradle setup, KMMBridge integration, CI workflows, signing configuration — well-trodden patterns AI handles confidently.
- Drift detection — Comparing the Android Kotlin and iOS Swift implementations of a feature for behavioral deltas was once careful diff-reading per feature; AI helps surface candidate deltas for human review.
- Documentation — Writing and maintaining migration documentation, PR descriptions, design notes, and ADRs.

Where AI doesn't help

- *Architectural seam decisions* (Chapter 4). AI suggests; humans decide. These decisions require organizational context AI doesn't have.

- *Team coordination and ownership* (Chapter 11). The hardest parts of a migration are organizational — who owns what, who reviews what, whose on-call handles what — and these are human problems.
- *Stage-gate approval cycles and release cadence.* Stages exist to manage risk, not code. AI does not change how often you can release safely.
- SDK availability — If an SDK has no KMP support, AI does not change that. The wrapper still has to be designed, maintained, and carry the SDK's behavior faithfully.
- Compliance and audit timelines — Regulatory clocks run at their own pace; AI speeds the writing of evidence, not the review of it.
- Device-testing time — Real devices, real OS versions, real human QA — AI accelerates test authoring, not test execution.
- Asymmetric-downside argument — The investment-preservation property of KMP is a structural argument that AI neither strengthens nor weakens.

Implications for the cost model and timeline

- Implementation vs. release cycles — AI speeds up raw implementation effort, but a phased migration still requires progression through multiple release cycles because each stage needs thorough review, team alignment, and production validation. While AI accelerates coding, the structural pacing set by team reviews, release schedules, and upskilling remains a key factor. When planning, consider estimating based on your specific team's release intervals across the migration stages rather than raw coding hours alone.
- Increasing human-system weight — The relative weight of human-system costs grows as mechanical costs fall, coordination, architectural-decision, and compliance costs become a larger share of total migration cost. Plan staffing to match — the lead-engineer and architect roles become *more* central, not less.
- Upskilling dynamics — Upskilling math changes. With AI as a reading and writing aid, productive contribution from an iOS engineer working in shared Kotlin can begin meaningfully sooner than the pre-AI baseline. Deep fluency — the ability to debug a Kotlin/Native crash, design a non-trivial `expect/actual` abstraction, or read an unfamiliar stack trace without assistance — still requires sustained exposure across the migration's lifecycle. Both are needed at different points; do not conflate them.
- Unchanged dealbreaker checklist — The dealbreaker checklist is unchanged. Nine of the ten dealbreakers are organizational, not implementation-effort related. AI does not turn a "no" into a "yes."

The baseline assumption

We assume your team uses modern AI coding assistants as standard tools. When this book cites a timeline, a cost, or a learning curve, the figure assumes the team is using AI as a baseline tool. The section above provides a leadership overview of what AI speeds up (and what it doesn't); Chapter 10 covers the day-to-day engineering practices.

If your organization restricts AI tooling for compliance or security reasons, revise the estimates upward — the magnitude depends on which restrictions, in which phases. We don't provide adjustment factors — the variance is too high to be useful. Talk to the engineers who would do the work.

> *For engineers*: the operational how-to of working with AI on a KMP/CMP migration — prompting patterns, failure modes, division of labor, day-to-day hygiene — lives in Chapter 10. The leader-facing summary above is the cost-model and budgeting argument; the engineering chapter is the practice.

Why KMP specifically?

This book is opinionated about KMP/CMP for production-app migrations. Six reasons matter for the decision:

- Direct platform access — KMP integrates at the language and build-tool level rather than introducing a runtime, bridge, or bytecode VM. Features with high platform affinity — VPN tunnels, background services, system permissions, hardware access, IPC, push, deep linking — keep direct access. There is no bridge layer to fight when the OS exposes a new capability.
- No JavaScript bridge or VM-style runtime — Shared Kotlin code compiles to Java bytecode for Android and to native binaries for iOS via Kotlin/Native. There is no JavaScript runtime, no bytecode VM, no framework bridge layer between the shared code and the platform. The Kotlin/Native runtime does add iOS binary weight (Chapter 9 quantifies this); what's absent is the bridge / VM overhead familiar from JS-based cross-platform stacks.
- Existing Kotlin expertise — If you ship Android, you ship Kotlin. KMP-shared code *is* Kotlin code — your existing Android engineers' skills transfer with no language switch. Switching cost is structurally lower than for cross-platform options that require a new primary language.
- Asymmetric adoption — iOS engineers do not need to learn Kotlin to participate. JetBrains explicitly recommends a model where Android engineers manage the Gradle/build side while iOS engineers initially consume the framework as if it were any other iOS library. Skill-up timing is a planning input, not a precondition.
- Lean dependency surface — The core stack is from JetBrains and platform vendors. There is no mandatory third-party UI runtime, no npm-style package ecosystem, no codegen pipeline between TypeScript and native. For organizations that operate in regulated or security-sensitive domains, the smaller dependency surface is a meaningful property in itself.
- App Store compliance — There is no App Store rejection risk from cross-platform tooling. A recurring executive concern: Apple's App Store Review Guideline 2.5.2 restricts dynamic code execution. KMP's iOS output is a regular iOS framework compiled to native binaries — no dynamic code, not in the guideline's scope. JetBrains addresses this explicitly.

JetBrains frames KMP's design philosophy as *"share code as needed to lower cost or risk without constraining product decisions"* — and recommends teams *"optimize for adaptability"* rather than

maximum reuse. Keep that framing in mind: the goal is not to share everything; it is to share what is worth sharing, where it lowers cost, while keeping native what is worth keeping.

If your situation is dominated by API-consuming, data-rendering features (low platform affinity), other cross-platform options may also be reasonable choices. If you have meaningful platform-affinity exposure, KMP becomes harder to replace.

The asymmetric downside (what if we're wrong?)

What if we adopt KMP and it doesn't deliver?

For KMP, the answer is favorable:

- Value of shared Kotlin — Shared Kotlin code remains valuable. Kotlin is Android's primary language. Code in `commonMain` is, with limited friction, code Android already wants.
- Low porting effort — Business logic ports back to native with minimal effort. A shared repository or use-case in Kotlin is, on Android, already a native component. On iOS, it can be re-implemented in Swift over time — but the *design* (interfaces, models, contracts) carries over.
- Incremental fallback — Incremental fallback is possible. You can stop advancing the migration at any point. Pausing at intermediate milestones—such as sharing only data repositories, sharing only ViewModels while keeping UI native, or sharing all UI screens while keeping navigation native—are all legitimate, stable terminal states. There is no all-or-nothing commitment. (See Chapter 3 for a detailed breakdown of these migration stages.)

This safety net is one of the strongest arguments for KMP—yet technical guides rarely mention it because it's a business risk argument, not a coding one. JetBrains comes close in their multiplatform overview, framing KMP's selective-sharing model as *"turning architecture into a flexible, evolving decision instead of a fixed commitment"* — a description that applies equally to "what we share" and "whether we keep going."

The implication runs through the rest of the book: every stage in Chapter 3 is designed so that *not advancing* is itself a coherent end state, not a failure mode. Chapter 10 names the anti-pattern of skipping the rollback / unwinding design.

When *not* to migrate

Migration is the wrong move *now* if any of the following apply:

1. Single small team and leading platform — If one platform is 90%+ of your engineering and the other lags, fix the lagging platform first; migration overhead doesn't pay back yet.
2. Lack of cross-platform drift — If the two apps feel like the same product to users today, the value of sharing is small.

3. Unresolved critical SDKs — A critical SDK on your roadmap has no KMP support and no wrapper budget. This is the most common dealbreaker in published retrospectives. Verify SDK availability before committing.
4. Organization learning curve — The engineering organization cannot absorb the learning curve. If teams are already over-committed on technical debt or feature load, adding migration is doubling-down on overload.
5. Risky release cadence — Release cadence is already at risk. If you cannot ship features on schedule today, a migration will not help; it will be blamed for the cadence problem and it will not be wrong.
6. Misaligned senior leadership — Senior leadership is not aligned. Migrations stall when engineering, product, and platform leaders are not all signed up for a sustained, cross-quarter commitment. Get alignment first.
7. Inability to staff iOS teams — Your iOS team cannot be staffed for cross-language work. If contractor-heavy iOS staffing or imminent attrition makes the upskilling investment lose value, the math doesn't work.
8. Increased compliance burden — Compliance or audit obligations would be increased materially. Some regulated environments have not yet evaluated KMP; if you would inherit a compliance review, scope that review before committing.
9. Roadmap pressure — Roadmap pressure is already 100%. If every quarter is fully committed for the next 18 months, migration cannot start; revisit later.
10. No appetite for host-shell rewrite — Sharing user interface components is straightforward (screens can mount inside native XML Fragments or UIKit view controllers), but migrating the core app navigation to shared code requires rewriting the platform host shells (e.g., retiring Android's XML NavGraph/Activity shell and iOS's `UINavigationController` in favor of a unified Compose Navigation setup). If your organization has no appetite for this host-shell rewrite, and you cannot accept a hybrid state where screens are shared but navigation remains native, the migration's final stages may not be worth the cost. This is a planning consideration, not a hard dealbreaker. Most teams in this position should plan to share UI elements while keeping native navigation as their terminal state, treating a shared navigation migration as a separately scoped project to decide on later. (We explore this migration path in detail in Chapter 3.)

If any of these is a yes, the answer for now is *no*. The list is designed for re-evaluation: adoption readiness changes — your organization that says no in 2026 may say yes in 2028.

Re-evaluation cadence

Migration readiness changes. Run the checklist again:

- Annually for organizations not yet committed.
- At each major roadmap-planning cycle for those still on the fence.
- When a major SDK on the dealbreaker list publishes KMP support.

- When team capacity opens up.

A “no” today is not a “no” forever. The dealbreaker list is the trigger for revisiting.

Further reading

Cited and recommended for deeper engagement with the *why-migrate* question:

- Cross-platform mobile development — KMP value proposition, layered sharing model, adoption-risk framing.
- Native and cross-platform app development — JetBrains’ own decision criteria for native vs. cross-platform.
- Multiplatform pain points and the cross-platform response — the eleven-pain-point list referenced at the opening of this chapter.
- Build a cross-platform application — current adoption framing and architectural principles.
- Reasons to try Kotlin Multiplatform — JetBrains’ enumerated benefits.
- Use cases and examples — organizational profiles that benefit most.
- Curated case-study index (Android + iOS) — for application-specific patterns.
- Introduce Kotlin Multiplatform to your team — skeptical-team objections and how to address them. Pairs with Chapter 11.